CALCUL MATRICIEL

Nicolas CHIREUX

1 Calcul matriciel et pivot de Gauss

Les différentes bibliothèques utilisées seront :
— numpy pour array, dot, eye importée sous l'alias np par import numpy as np
— numpy.linalg pour matrix_power, solve et inv

1.1 Les matrices en python pur et avec numpy

Une matrice n'est qu'une liste de listes. Par exemple $A = [[1, 2], [3, 4]]$ représentera pour nous la matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$.

Les entrées de A sont obtenues via $A[i][j]$.

Noter que l'affichage d'une matrice n'est pas bien joli. Heureusement la bibliothèque numpy est là : $print(np.array(A))$ arrange tout ça et donne une présentation plus usuelle.

Attention : En python, les indices commencent à 0, donc pour les matrices à $A[0][0]$.

Une liste est un objet mutable. On peut modifier les entrées de la matrice, c'est bien, les fonctions peuvent transformer la matrice qu'elles ont en argument, c'est bien, mais il faut prendre garde aussi aux problèmes des éventuelles copies d'une matrices qui pointeraient vers la même case mémoire... une modification de la copie modifiera aussi l'original. On rappelle qu'il existe un module appelé copy qui résout ce problème des copies.

1.2 Opérations sur les matrices

Le but ici est de programmer diverses opérations en python pur puis de comparer avec les mêmes fonctions intégrées à numpy quand celles-ci existent.

1. Écrire une fonction **matrice_nulle** prenant en entrée deux entier (n, m) et renvoyant la matrice nulle de taille nxm.
2. Écrire une fonction **dimension** prenant en entrée une matrice et renvoyant le nombre de lignes et le nombre de colonnes.
   Indication : on pourra utiliser l'instruction *len* qui donne la taille d'une liste.
3. Écrire une fonction **addition** prenant en entrée deux matrices (de même taille !) et renvoyant leur somme.
   Comparer avec la version utilisant numpy : $np.array(A) + np.array(B)$
4. Écrire une fonction **transposee** prenant en entrée une matrice et renvoyant sa transposée.
5. Écrire une fonction **multiple** prenant en entrée une matrice A et un scalaire $coef$ et renvoyant la matrice $coef.A$.
   Comparer avec la version utilisant numpy : $np.array(A) * coef$
6. Écrire une fonction **multiplication** prenant en entrée deux matrices de tailles compatibles (n, p) et (p, m) par exemple) et renvoyant leur produit.
   Comparer avec la version utilisant numpy : $np.dot(np.array(A), np.array(B))$
7. Écrire une fonction **puissance** prenant en entrée une matrice A et un entier $n \geq 0$ et renvoyant la matrice A^n
   Comparer avec la version utilisant numpy : $np.linalg.matrix\_power(np.array(A), n)$

Évaluer la complexité des opérations **addition**, **transposee**, **multiple**, **multiplication** et **puissance**.

1.3 Pivot - Mise sous forme triangulaire d'une matrice

1.3.1 Position du problème

On considère un système $Y = TX$ avec $T \in TS_n(\mathbb{K})$ - matrice triangulaire - inversible et $Y \in M_{n,1}(\mathbb{K})$ fixés, et on cherche l'unique solution $X \in M_{n,1}(\mathbb{K})$ de ce système.

Un tel système se résout de bas en haut. A la ligne L_n, on va seulement faire un quotient pour trouver $x_n = \frac{y_n}{t_{n,n}}$. Mais ensuite à la ligne L_i, on obtiendra x_i par :

$$x_i = \frac{1}{t_{i,i}} \left(y_i - \sum_{j>i} t_{i,j} x_j \right) \tag{1}$$

1.3.2 Mise sous forme triangulaire

On fixe une matrice $A \in M_n(\mathbb{K})$ inversible.

Pivot à la première étape Au début de la méthode du pivot, on veut "nettoyer" la première colonne en utilisant $A_{1,1}$ comme pivot, via $L_i \leftarrow L_i - \mu_i L_1$ pour $i \geq 2$ où $\mu_i = \frac{A_{i,1}}{A_{1,1}}$. Mais un des problèmes est que $A_{1,1}$ peut être nul.

On doit donc d'abord rechercher la première entrée non nulle dans la colonne 1 et une fois cette entrée trouvée à une ligne L_i échanger L_i et L_1. Comme A est inversible, on sait qu'on va toujours en trouver, car la première colonne ne peut pas être nulle.

Rem : Une précaution liée au calcul numérique pour le choix du pivot :

Quand on manipule des flottants, le test de l'égalité à zéro n'est pas adapté, car parfois dans les calculs un nombre qui vaudrait théoriquement zéro sera remplacé par un nombre très petit. On pourrait remplacer le test d'égalité à zéro par la comparaison avec le epsilon machine, mais pour des raisons de meilleure précision du calcul numérique,il vaut mieux éviter de diviser par des très petits nombres. Il vaut donc mieux choisir comme pivot dans la colonne 1, l'entrée ayant la plus grande valeur absolue.

Pivot à l'étape i A l'étape i, en suivant le principe précédent, on choisit parmi les lignes L_j avec $j \geq i$, celle où l'entrée $A_{j,i}$ a la plus grande valeur absolue, et on l'échange avec L_i. Ensuite on se sert de cette nouvelle L_i pour nettoyer la colonne C_i en dessous de la diagonale.

Rem : là encore, on est sûr qu'une des entrées $A_{j,i}$ avec $j \geq i$ est non nulle. Sinon, comme à ce stade les premières colonnes $C_1 \dots C_{i-1}$ sont déjà celles d'une matrice T.S., on aurait $C_1 ... C_i$ liées.

On a donc l'algorithme suivant :

```
A : matrice
Pour i de 0 à n − 2 faire
    trouver j ≥ i tel que |A_{j,i}| soit maximum
    échanger L_i et L_j
    Pour k de i + 1 à n faire
        L_k ← L_k − μ_k L_i
    Fin Pour
Fin Pour
Renvoyer A
```

Algorithme 1: Mise sous forme triangulaire d'une matrice A

Complexité de la méthode Pour chaque valeur de $i \in [n-2]$:

— la recherche de j coûte $n-i$ comparaisons

— l'échange éventuel L_i et L_j coûte $2n+2$ affectations : $2n$ pour les entrées des lignes et 2 pour i et j.

— pour chaque valeur de k entre $i+1$ et $n-1$, la transvection coûte n affectations, et autant de divisions, multiplications, soustractions (une par entrée de la ligne).

En ajoutant toutes ces contributions comme comptant 1, on obtient pour la méthode du pivot sur une matrice $A \in GL_n(\mathbb{K})$ un coût en $\Theta(n^3)$.

Travail à effectuer

1. Écrire une fonction **echange_ligne(A,i,j)** prenant en argument la matrice A et échangeant les lignes i et j. Attention, cette fonction ne renvoie rien : elle modifie en place la matrice donnée en argument.
2. Écrire une fonction **transvection(A,i,j,mu)** prenant en argument la matrice A et calculant $L_i \leftarrow L_i + mu.L_j$ Attention, cette fonction ne renvoie rien : elle modifie en place la matrice donnée en argument.
3. Écrire une fonction **pivot_partiel(A, j0)** chargée de la recherche du pivot de module maximal dans une matrice sur une colonne $j0$ à partir de la ligne $j0$: il s'agit de renvoyer le (un) $i \geq j0$ tel que $|A_{i,j0}|$ est maximal.
4. Écrire une fonction **devient_triangle(A)** qui met sous forme triangulaire la matrice A. Cette fonction renvoie la matrice triangularisée.

1.4 Résolution d'un système de Cramer

Définition : Un système de Cramer est un système $Y = AX$ avec A matrice carrée inversible données, Y colonne donnée, et d'inconnue une colonne X. On sait que ce système admet une unique solution.

Propriété : Soit un système linéaire quelconque $S : Y = AX$ avec $A \in M_{m,n}(\mathbb{K})$, $Y \in M_{m,1}(\mathbb{K})$ et $X \in M_{n,1}(\mathbb{K})$ où A et Y sont fixés et on cherche X. Si on transforme ce système S par des opérations élémentaires sur les lignes, on arrive à un système équivalent $S' : Y' = A'X$ où Y' et A' ont été modifiées par les mêmes opérations sur les lignes. Ces systèmes équivalents ont un même ensemble de solutions.

Première étape de la résolution Transformer A en une matrice A' triangulaire supérieure avec la méthode de triangulation par pivot précédente.

Appliquer les mêmes transformations sur les lignes de Y ce qui donne une nouvelle colonne Y'.

Rem : pour ceux qui préfèrent, on peut travailler sur la matrice augmentée $(A|Y)$ obtenue en rajoutant Y comme dernière colonne à la matrice A. Cela évite de dupliquer les lignes de code mais en terme de nombre d'opérations, ça ne change rien.

Deuxième étape de la résolution Le système est sous forme triangulaire :

$$\begin{cases} a'_{1,1}x_1 + a'_{1,2}x_2 + \ldots + a'_{1,n-1}x_{n-1} + a'_{1,n}x_n &= y'_1 \\ a'_{2,2}x_2 + \ldots + a'_{2,n-1}x_{n-1} + a'_{2,n}x_n &= y'_2 \\ \ldots & \\ a'_{n-1,n-1}x_{n-1} + a'_{n-1,n}x_n &= y'_{n-1} \\ a'_{n,n}x_n &= y'_n \end{cases} \tag{2}$$

Il n'y a plus qu'à "remonter", via des substitutions. Il s'agit donc de calculer :

$$x_i = \frac{1}{a'_{i,i}} \left(y'_i - \sum_{k=i+1}^{n-1} a'_{i,k} x_k \right) \tag{3}$$

Écrire une fonction **resolution_systeme(A, y)** qui résolve le système de Cramer $Y = AX$. Cette fonction renvoie la matrice colonne X.

Comparer avec la version utilisant numpy : $np.linalg.solve(np.array(A), np.array(B))$

Complexité de la résolution par pivot Cherchons la complexité de la résolution :

— Mise sous forme triangulaire (déjà vue plus haut)
- $(n-1)+(n-2)+\ldots+1$ transvections soit n^2 transvections, soit n^3 opérations flottantes
- les recherches de maximum :n^2 tests

— Remontée : n^2 additions, soustractions, multiplications ou quotients de flottants.

Pour la méthode du pivot sur une matrice $A \in GL_n(\mathbb{K})$, la résolution à un coût en $\Theta(n^3)$. La remontée ne change pas la complexité.

1.5 Inversion de matrice

L'algorithme du pivot nous a amené à faire des opérations sur les lignes d'une matrice, ce qui revient à la multiplier par des matrices élémentaires que nous appellerons $P_1, P2, \ldots, P_N$ (après N opérations).

Le système $AX = Y$ est alors équivalent à $P_1AX = P_1Y$ puis $P_2P_1AX = P_2P_1Y$, puis

$$P_N \ldots P_2P_1AX = P_N \ldots P_2P_1Y \Leftrightarrow X = P_N \ldots P_2P_1Y \tag{4}$$

car si on a correctement appliqué l'algorithme du pivot, le membre de gauche vaut exactement X.

Ainsi : $P_N \ldots P_2P_1 = A^{-1}$. Cette matrice est en fait exactement celle obtenue en appliquant à la matrice identité (et dans le même ordre) les opérations réalisées sur A.

En appliquant cette idée, écrire une fonction **inversion(A)** qui calcule l'inverse de la matrice A. Cette fonction renvoie la matrice A^{-1}.

Comparer avec la version utilisant numpy : $np.linalg.inv(np.array(A))$